INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Polychronous Design of Embedded Real-Time Systems

Abdoulaye GAMATIÉ — Thierry GAUTIER — Paul LE GUERNIC — Jean-Pierre TALPIN

N° 5509

Mars 2005

THÈME 1

Rapport de recherche

N 0249-6399 ISRN INRIA/RR--5509--FR+ENG

Polychronous Design of Embedded Real-Time Systems

Abdoulaye Gamatié , Thierry Gautier , Paul Le Guernic , Jean-Pierre Talpin

Thème 1 — Réseaux et systèmes
Projet Espresso

Rapport de recherche n° 5509 — Mars 2005 — 42 pages

Abstract: This report proposes a design methodology for embedded real-time systems using a synchronous multi-clocked framework, which provides a well-defined mathematical model that yields rigorous methodological support for the trusted design, validation and automatic code generation. The presented methodology addresses among others the non trivial issue of modeling asynchronous mechanism using the synchronous paradigm. Among target application domains, we mainly focus on the avionics area. A library of polychronous models of services has been implemented, based on the avionic standard APEX-ARINC 653. These services describe functionalities of a real-time operating system in integrated modular avionics architectures. The proposed methodology has been applied to some case studies in avionics. The library is also the basis for another study that consists of modeling Real-Time Java applications in the polychronous framework. A great advantage is that formal transformation and optimization techniques become enabled. Finally, a major contribution of the work exposed in this report is the convergence between the theory of formal methods, industrial practice and current trends in embedded real-time system design.

Key-words: Design methodology, synchronous approach, polychrony, embedded real-time systems, Avionics, formal methods, Signal

The present work has been partially supported by the European project IST SafeAir (Advanced Design Tools for Aircraft Systems and Airborne Software) [GMGW01] and the French National Network on Software Technologies (RNTL) project Expresso.

Conception polychrone de systèmes embarqués temps réel

Résumé : Ce rapport propose une méthodologie de conception pour les systèmes embarqués temps réel dans un cadre synchrone multi-horloge. Ce dernier offre un modèle mathématique bien défini qui fournit un support méthodologique rigoureux pour la conception fiable, la validation et la génération automatique de code. La méthodologie présentée aborde notamment la question non triviale de la description de mécanismes asynchrones à l'aide du paradigme synchrone. Parmi les domaines visés, nous nous appuyons sur l'avionique. Une bibliothèque de modèles polychrones de services a été mise en œuvre, basée sur le standard avionique APEX-ARINC 653. Ces servives décrivent les fonctionnalités d'un exécutif temps réel dans les architectures dites modulaires intégrées. La méthodologie proposée a été appliquée à des cas d'étude du domaine de l'avionique. La bibliothèque a également servi de base pour une autre étude qui consiste à modéliser, dans le cadre polychrone, des applications écrites en Java temps réel. Un gros avantage réside dans le fait que les techniques de transformations formelles et d'optimisation deviennent applicables. Enfin, un apport non négligeable du travail présenté dans ce rapport consiste à faire effectivement converger la théorie des méthodes formelles, la pratique industrielle et les tendances actuelles dans la conception des systèmes embarqués temps réel.

Mots-clés : Méthodologie de conception, approche synchrone, polychronie, systèmes embarqués temps réel, avionique, méthodes formelles, SIGNAL

Contents

1 Introduction

Embedded real-time systems are devices consisting of hardware and software with functional and timing constraints for the interaction with their environment. They are characterized in today's technologies by their pervasiveness and ubiquitousness. Typical domains where embedded real-time systems are encountered are telecommunication, nuclear power plants, avionics, and medical technology. These systems are often critical because of the high human and economic stakes. Therefore, the development of such systems requires highly reliable methodologies. Over the past decade, high-level design of such systems has gained prominence in the face of rising technological complexity, increasing performance requirements, and shortening time to market demands. Broad discussions of challenges in the design of these systems can be found in literature [Lee00] [Wir01] [Sif01] [Pnu02]. It is now widely accepted that the suitable design frameworks must provide a means to describe systems without ambiguity, to check desired properties of these systems, and to automatically generate code with respect to requirements.

The *synchronous approach* has been proposed in order to answer this demand [BCE$^+$03]. Its basic assumption is that computations and communications are instantaneous. This assumption is often represented by the ideal vision of *zero time* execution (referred to as "synchronous hypothesis"). More precisely, a system is viewed through the chronology and simultaneity of observed events during its execution. This is a main difference from classical approaches where the system execution is rather considered under its chronometric aspect (i.e., duration has a significant role). The mathematical foundations of the synchronous approach provide formal concepts that favor the trusted design of embedded real-time systems.

The multi-clocked or *polychronous* model [LTL03] stands out from other synchronous specification models by its capability to allow the design of systems where each component holds its own activation clock as well as single-clocked systems in a uniform way. A great advantage is for example its convenience for component-based design approaches and a modular development of modern systems, the complexity of which is rapidly growing. As a matter of fact, the design of each component can be addressed separately. The expressiveness of its underlying semantics allows to deal with several issues on real-time design.

On the other hand, the popular slogan "*write once, run anywhere*" effectively renders the expressive capabilities of general purpose programming languages for developing, deploying, and reusing target-independent applications. Generality and simplicity has driven most attention of the compiler technology community to developing local and compositional compiler optimization techniques. When it comes to the implementation of embedded software, this approach is however far from satisfactory, especially in hard real-time system design (e.g. airborne systems, digital circuits) where conformance to real-time specifications is critical.

Domain-specific models and languages, such as these proposed under the synchronous programming paradigm, provides the necessary formal engineering models and design methodologies to allow for a program written once to be mapped on any distributed execution architecture by using global transformation and optimization techniques.

In this article, we consider the polychronous model for the definition of a methodology in order to address the design of embedded real-time systems. The central idea of this methodology is based on a partitioning of a system into two levels:

- *functional level*: its includes descriptions of both *applications architecture* (e.g. decomposition of a program into threads and how these threads are grouped into processes) and *applications functionalities* (e.g. the threads of a program associated with an application periodically or sporadically react to inputs from the environment by interacting with each other for the access to shared data).
- *execution architecture level*: depending on the needed detail level, it can be a high level description of a hardware platform, a middle-ware library, a real-time operating system, a virtual machine (e.g. in Java), a simulation kernel (e.g. in SystemC).

The issues resulting from the above decomposition can now be considered in specific ways:

- *modeling*: the execution architecture, considered through an application programming interface (API) of generic services, is modeled by template propositions. For instance, the procedure for thread creation in an API associated with a real-time operating system (RTOS) corresponds to a template proposition in the RTOS model whose parameters are the number of threads supported by the application scheduler, the period and deadline of the thread (for a real-time thread), etc.
- *analysis*: the application architecture, considered as a hierarchical structure, is interpreted to elaborate a model by the instantiation of generic API services to the parameters and initial values provided in the program (e.g. thread parameters).
- *translation*: each thread consists of a sequential program that describes a functionality to be periodically or sporadically executed by the scheduler and corresponds to a particular model.

This allows for a complete separation of the virtual (threading or functional) architecture of an application from its actual, real-time and resource-constrained implementation: it provides an implementation of the "*write once run anywhere*" slogan in embedded system design.

The methodology proposed in this article arises from previous work on real-time operating systems modeling, embedded systems modeling and verification in the POLYCHRONY workbench[1], a tool-set for embedded system design based on a multi-clocked synchronous model

[1] URL: `http://www.irisa.fr/espresso/Polychrony`

of computation and implemented by the data-flow notation SIGNAL [BLJ91]. In [GG03], we describe the implementation of a real-time operating system standard for avionic applications: APEX-ARINC [Air97b]. A commercial implementation of this library, RT-Builder from TNI-Valiosys[2], is used for industrial-scale embedded software engineering project in avionics. In [TGB+03], we use the academic version of this library (defined within POLYCHRONY) to describe key services of the Real-Time Java virtual machine. It is applied to rethreading multi-threaded real-time Java programs by global optimization. In [TLS+04], the application of our methodology to system-level design is further developed by studying its application to checking behavioral conformance between embedded systems described in SpecC and at heterogeneous levels of abstraction. Finally, in [TBS+04], a generic translation scheme of SystemC programs to the POLYCHRONY workbench is described by considering a static single assignment intermediate representation due to the Gnu-CC project.

Contribution. The first contribution of the work presented in this article is the design of embedded real-time systems using the polychronous model. These systems often include asynchronous mechanisms (e.g. to achieve communications). The modeling of such mechanisms using the synchronous paradigm is not trivial. A methodology is proposed within a homogeneous framework where the unique formalism is the polychronous language SIGNAL. As a result, the formal techniques and tools available within the development platform associated with SIGNAL, called POLYCHRONY, can be used at each step for system analysis. Targeting avionics, we particularly focus on the recent *integrated modular avionics* architectures and their associated standard ARINC [Air97a] [Air97b]. We develop a library of so-called APEX-ARINC services, which provide polychronous models of RTOS functionalities. On the other hand, we show how synchronous design tools allow to model embedded software written in a high-level and general purpose programming language such as Real-Time Java. This adds a formal engineering model and methodology, allowing to adhere to hard real-time constraints while offering at the same time profound transformation and optimization techniques.

Outline. The rest of the report is organized as follows: Section 2 exposes some relevant works in the field of embedded real-time systems. Then, Section 3 introduces the polychronous design language SIGNAL, its associated constructs and analysis techniques. Section 4 first gives an overview of our design methodology for embedded real-time systems. A few application examples studied recently are reported in Sections 4.2 and 4.3. Finally, in Section 5, we give conclusions.

2 Related work

Several design approaches have been defined for embedded real-time systems. Here, we mention those that could be considered to be representative for the different families of ap-

[2] URL: `http://www.tni-valiosys.com`

proaches that are related to the one exposed in this paper.

The first approach we mention is TAXYS [CPP+01]. It has been proposed for the design and validation of real-time embedded applications. It is partly based on the synchronous approach by using the synchronous language ESTEREL with an associated compiler. The model checker KRONOS is used for timing and schedulability analysis. So, TAXYS combines synchronous approach and timed automata theory.

The GIOTTO approach [HHK01] considers a specific language for the design that is used to define abstract models of embedded control systems. The language has a time-trigger semantics that facilitates time predictability for system analysis. The associated compiler and a runtime library enable the implementation on a target platform. One interesting common characteristic of the GIOTTO formalism and the synchronous ones is that they both allow platform-independent specifications.

Component-based design techniques have revealed advantages, re-usability being the most prominent. We mention two approaches: METAH [Ves97] and VEST [SZP+03]. In the former, the design activity relies on the architecture description language *MetaH*. The approach addresses embedded real-time, distributed avionic applications. It proposes a toolset for the description and combination of software and hardware components to construct a system. Facilities are also provided for real-time analysis. The VEST approach is similar with METAH. It also provides a tool that supports infrastructure creation, embedded system composition, and mapping of passive software components to active run-time structures (such as tasks). A set of analysis tools is also available for real-time and reliability analysis. To enable this, a library of micro-components, components, and infrastructures has been defined. Micro-components are passive software such as interrupt handlers, dispatchers, and plug and unplug primitives. The tool supports dependency checks and composition. However, VEST does not support formal proof of correctness.

Other approaches consider a separation of concerns in system design. The SYNDEX approach [GS03] separates aspects inherent to an application from those related to the implementation platform. SYNDEX proposes a methodology called *algorithm architecture adequation* (AAA) supported by system-level Computer-Aided Development software called for the design of distributed embedded real-time systems. The AAA methodology wholly covers design steps: from the specification of functionalities (i.e. the software level) to the deployment on target multi-processor architectures including specific integrated circuits (i.e. the hardware level). Basically, this is achieved using a graphical environment, which allows the designer to manually and/or automatically identify efficient ways to distribute application functionalities on the target architecture (the criterion is to satisfy timing requirements and to minimize hardware resources). The distribution relies on graph representations, and takes non-functional requirements inputs (e.g. real-time performances). Finally, a possibility of automatically generating code corresponding to the specified system is offered. In addition to SynDEx, we can briefly mention the AutoFocus approach [Rom02], which follows similar ideas. It addresses distributed system design by offering integrated hierarchical description techniques for different views of a systems: the structure of a system (its components and communication between them), its behavioral description and the way the system interacts with its environment.

Among approaches dedicated to the design of embedded real-time systems, only a few of them address both continuous and discrete activities of such systems. The approach based on the Charon language [ADE+03] enables modular specifications of interacting ***hybrid systems*** based on the notions of agent (a building block) and mode (an hierarchical state machine). The description of a system follows two level hierarchy: the architectural level provides the structure of the system in terms of composed agents whereas the behavioral level indicates the interaction modes.

Finally, we mention the Ptolemy approach [Lee01], which supports the modeling, simulation, and design of embedded systems. It integrates several computation models (e.g. synchronous/reactive systems, continuous time, etc.) in order to deal with concurrency and time. A key point is that the designer can simulate different kinds of interactions between active components. Therefore, the focus is mainly on the choice of suitable computation models for an appropriate type of behavior in the system.

In the next section, we introduce the Signal language, which relies on the polychronous model. We first describe its basic objects and constructs. Then, we briefly discuss the analysis of Signal programs.

3 The polychronous design language Signal

Signal [BLJ91] handles unbounded series of typed values $(x_t)_{t\in\mathbb{N}}$, denoted as `x`, implicitly indexed by discrete logical time (denoted by t in the semantic notation) and called *signals*. At a given instant, a signal may be present, then it holds a value; or absent, then it is denoted by the special symbol $\perp$ in the semantic notation. There is a particular type of signals called `event`. A signal of this type is always *true* when it is present. The set of instants where a signal `x` is present is called its *clock*. It is noted as `^x` (which is of type `event`) in the language. Signals that have the same clock are said to be *synchronous*. A Signal program, also called *process*, is a system of equations over signals.

3.1 Signal constructs

Signal relies on a handful of primitive constructs, which can be combined using a composition operator. These core constructs are of sufficient expressive power to derive other constructs for comfort and structuring. Here, we give a sketch of the primitive constructs (bold-faced) and a few derived constructs (italics) often used. For each of them, the corresponding syntax and definition are mentioned.

Functions/Relations: `y:= f(x1,...,xn)` $\stackrel{def}{=}$ $y_t \neq \perp \Leftrightarrow x1_t \neq \perp \Leftrightarrow ... \Leftrightarrow xn_t \neq \perp$, $\forall t$: $y_t = f(x1_t, ..., xn_t)$.
Delay: `y:= x $ 1 init y0` $\stackrel{def}{=}$ $x_t \neq \perp \Leftrightarrow y_t \neq \perp$, $\forall t > 0$: $y_t = x_{t-1}, y_0 = y0$.
Down sampling: `y:= x when b` $\stackrel{def}{=}$ $y_t = x_t$ if $b_t = true$, else $y_t = \perp$. The derived statement `y:= when b` is equivalent to `y:= b when b`.
Deterministic merging: `z:= x default y` $\stackrel{def}{=}$ $z_t = x_t$ if $x_t \neq \perp$, else $z_t = y_t$.
Parallel composition: `(| P | Q |)` $\stackrel{def}{=}$ union of equations associated with `P` and `Q`.
Hiding: `P where x` $\stackrel{def}{=}$ `x` is local to the process `P`.

Synchronization: `x1 ^= x2` $\stackrel{def}{=}$ `(| h := (^x1 = ^x2) |) where h`.
Clock union: `h:= x1 ^+ x2` $\stackrel{def}{=}$ `h:= ^x1 default ^x2`.
Memory: `y:= x cell b init y0` $\stackrel{def}{=}$
`(| y := x default (y$1 init y0) | (y ^= x ^+ (when b) |)`.

SIGNAL also provides a process construct in which any process may be "encapsulated" (see example on FIG. 7). This allows to abstract a process to an interface so that the process can be used afterwards as a black box through its interface, which describes the input-output signals and parameters. The process frame enables the definition of sub-processes. Finally, put together, all these features of the language favor modularity and re-usability.

3.2 Analysis of SIGNAL programs

Functional properties. They consist of *invariant properties* on the one hand (e.g. a program exhibits no contradiction between clocks of involved signals), and *dynamic properties* on the other hand (e.g. reachability, liveness). The SIGNAL "compiler" itself addresses only invariant properties (here, the compiler includes more functionalities than classical compilers. For example, in addition to standard syntax or type checking, it implements static verification algorithms and allows for automatic generation of the optimized code). For a given SIGNAL program, the compiler checks the consistency of constraints between signal clocks, and statically proves properties (e.g. determinism, absence of cyclic definitions, absence of empty clocks to ensure a consistent reactivity of the program). A major part of the compiler task is referred to as the *clock calculus*. Dynamic properties are often addressed using the model checker SIGALI [MBLL00].

Non functional properties. They include *temporal properties*, which are of high interest for real-time systems. A technique has been defined in order to allow timing analysis of SIGNAL programs [KL96]. Basically, it consists of formal transformations of a program into another SIGNAL program that corresponds to a *temporal interpretation* of the initial one. The new program serves as an *observer* of the initial program. An observer of a program `P`

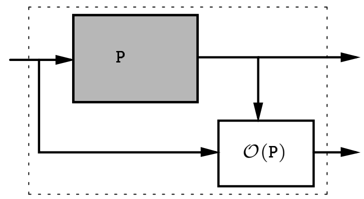


Figure 1: Composition of a program P together with its observer $\mathcal{O}$(P).

is an *abstraction* $\mathcal{O}$(P) of P in which we only specify the properties we want to check. The term "abstraction" means here that $\mathcal{O}$(P) does not constrain the original behavior of P when the two programs are composed. As shown in Figure 1, the observer receives from the observed program the signals required for analysis and indicates whether or not the considered properties have been satisfied (this can be expressed, e.g., through boolean output signals like in LUSTRE programs [HLR93]). The use of observers for verification is very practical because they can be easily described in the same formalism as the observed program. Thus, there is no need to combine different formalisms as in other analysis techniques such as some model-checking techniques, which associate temporal logics with automata [DY95].

Finally, the POLYCHRONY workbench implements a multi-clocked synchronous model of computation (the polychronous MoC [LTL03]) to model control-intensive embedded software using the SIGNAL language and to support a formal, refinement-based, design methodology [TLS+04] with companion decision procedures to validate key design steps using precise formal design properties (e.g. controllability of a component by its environment, invariance of a design refinement under flow-equivalence) and/or automatic design transformations and refinement algorithms (control hierarchization, protocol synthesis).

4 A design methodology

4.1 Global view

A major part of design environments encompasses multiple tools for various purposes: specification, verification, evaluation, etc. While such environments are practical for the design of large systems, a drawback is that it is difficult to guarantee the correctness of systems. As a matter of fact, the involved tools have in general different semantic foundations. This leads to a global coherence problem, which affects the description, verification or validation of the systems and represents a big obstacle to the development of safety critical systems such as embedded real-time systems. One solution to this problem consists in considering the same semantic model for the all design activity.

The inherent flexibility of the abstract notions defined in the polychronous framework invites and favors the design of correct-by-construction systems by means of well-defined transformations of system specifications that preserve the intended semantics and stated properties of the system under design.

In our design methodology, the logical description of the functionalities of a system is kept separate from the description of the other parts of the system, which depend on the underlying operating system and hardware. In fact, the idea adopted here shares some similarities with the one considered in SynDEx [GS03] or AutoFocus [Rom02]. Therefore, it results two levels: *functional level* and *execution architecture level.* An immediate consequence of such a hierarchical decomposition is that the logical model of the system is retargetable.

Functional level. At this level, we adopt an approach that is based on description refinements. The formal semantics of the polychronous model leads to the definition of semantic equivalence relations. These relations are used to guarantee that transitions from one description of a system to another description preserves the original semantics of the system. For instance, taking into account timing scalability during refinements amounts to check the so-called *stretch-equivalence* of original behaviors and the resulting ones [LTL03]. The desynchronization of behaviors from a given description and an associated refinement can be checked using flow-equivalence (intuitively, two behaviors are flow-equivalent if the values associated with their common variables are observed in the same order [LTL03]; synchronization relations may differ). In practice, the compiler of Polychrony allows to perform refinements of Signal programs and guarantee correct-by-construction models of a system. The performed transformations rely on a *hierarchical conditional dependency graph* (HCDG) [ABL95] which canonically represents a program. This graph contains information required for the transformations (e.g. clock information and dependencies between signals).

Execution architecture level. The target architecture is composed of a set of possibly heterogeneous execution components such as processors and micro-controllers. A general observation is that the level of detail at which the architecture needs to be known depends on the refinement of the mapping to the chosen architecture. This means that in the simplest cases the amount of data required is fairly small and simple to assess:

- the set of processors or tasks, and the mapping from operations or sub-processes in the application specification to those processors or tasks. This information enables the partitioning of the graph into sub-graphs grouped according to the mapping.
- the topology of the network of processors, the set of connections between processors, and a mapping from inter-process communications to these communication links. This is useful in the case of signals exchanged between processes located on different processors or tasks, if several of them have to be routed through the same communication medium.

- a definition of the set of system-level primitives used e.g. for communications (read/write to the media). Roughly, this amounts to the profiles of the function library to which the code generated for the application has to be linked.

Further degrees of refinement of the description may be required for a better architecture-adaptation: for example, concerning communications, the type and nature of the links (that could be implemented using shared variables, synchronous or asynchronous communications). If the target architecture features an operating system (OS), the required model consists basically in the profile of the corresponding functions. For instance, according to the degree of use of the OS, we need models of synchronization gates, communications (possibly including routing between processors) or task management services (e.g. *start*, *stop*, *suspend* and *resume*).

4.2 Polychronous design of avionic systems

4.2.1 Generalities on avionic architectures

Traditional architectures in avionic systems are called *federated* [Air97a] [Rus99]. Functions with different criticality levels are hosted on different fault-tolerant computers. Fig. 2 illustrates such an architecture where n functions are considered. A great advantage of those architectures is fault containment. However, this potentially leads to high risks of massive usage of computing resources since each function may require its dedicated computer. Consequently, maintenance costs can increase rapidly.

Figure 2: Federated architectures: each avionics function has its own fault-tolerant computers.

Integrated Modular Avionics (IMA). The recent IMA architectures propose a new way to deal with major obstacles inherent to federated architectures [Air97a]. In IMA, several functions with possibly different criticality, are allowed to share the same computer resources (see Fig. 3). They are guaranteed a safe allocation of shared resources so that no fault propagation occurs from one component to another component. This is achieved through a *partitioning* of resources with respect to available time and memory capacities.

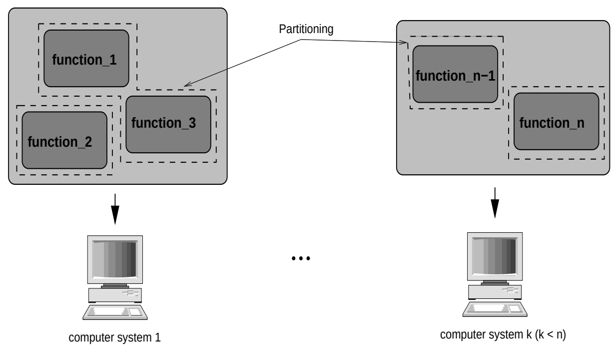


Figure 3: Integrated Modular Avionics: different functions can share a fault-tolerant computer.

A *partition* is a logical allocation unit resulting from a functional decomposition of the system. IMA platforms consist of *modules* grouped in *cabinets* throughout the aircraft. A module can contain several partitions that possibly belong to functions of different criticality levels. Mechanisms are provided in order to prevent a partition from having "abnormal" access to the memory area of another partition. A processor is allocated to each partition for a fixed time window within a major time frame maintained by the *module-level OS*. A partition cannot be distributed over multiple processors either in the same module or in different modules. Finally, partitions communicate asynchronously via logical *ports* and *channels*. Message exchanges rely on two transfer modes: *sampling* mode and *queuing* mode. In the former, no message queue is allowed. A message remains in the source port until it is transmitted via the channel or it is overwritten by a new occurrence of the message. A received message remains in the destination port until it is overwritten. A refresh period attribute is associated with each sampling port. When reading a port, a *validity* parameter indicates whether the age of the read message is consistent with the required refresh period attribute of the port. In the queuing mode, ports are allowed to store messages from a source partition in queues until they are received by the destination partition. The queuing discipline for messages is First-In First-Out (FIFO).

Partitions are composed of *processes* that represent the executive units[3]. Processes run concurrently and execute functions associated with the partition in which they are contained. Each process is uniquely characterized by information (like its period, priority, or deadline time) useful to the *partition-level OS* which is responsible for the correct execution of processes within a partition. The scheduling policy for processes is priority preemptive.

[3]An IMA partition/process is akin a UNIX process/task.

Communications between processes are achieved by three basic mechanisms. The bounded *buffer* allows to send and receive messages following a FIFO policy. The *event* permits the application to notify processes of the occurrence of a condition for which they may be waiting. The *blackboard* is used to display and read messages: no message queues are allowed, and any message written on a blackboard remains there until the message is either cleared or overwritten by a new instance of the message. Synchronizations are achieved using a *semaphore*. Figure 4 illustrates an example of IMA partitioning.

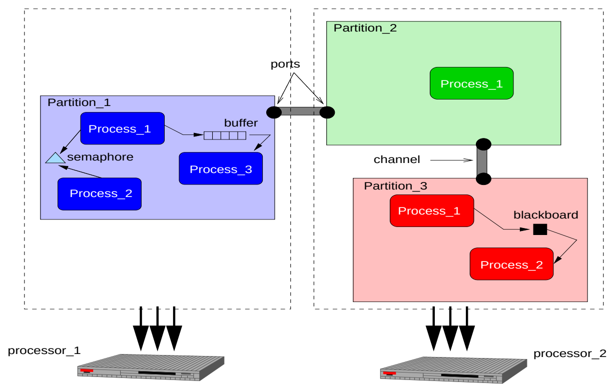


Figure 4: An example of application partitioning.

Several standards for software and hardware have been defined for IMA. Here, we particularly concentrate on the APEX-ARINC 653 standard [Air97b], which proposes an OS interface for IMA applications, called *Avionics Application Software Standard Interface* (see Fig. 5). It includes services for communication between partitions on the one hand and between processes on the other hand, synchronization services for processes, partition and process management services, and time and error management services.

4.2.2 Polychronous design: basic building blocks

The polychronous design of avionic applications relies on a few basic blocks [GG03], which allow us to model partitions:

1. APEX-ARINC 653 services (they describe communication and synchronization, partition, process and time management...);
2. an RTOS model (it is partially described using complementary services providing functionalities that are not provided via APEX-ARINC 653 services, such as process scheduling);

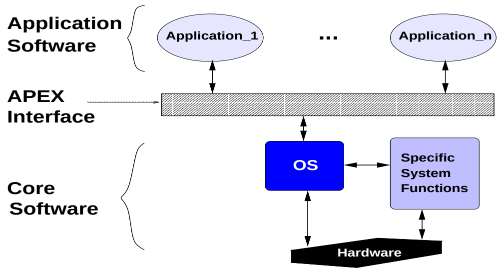


Figure 5: The APEX interface within the core module Software.

3. executive entities (they mainly consist of generic partition and process models).

In the following, we show for each building block, the way its corresponding SIGNAL model is obtained.

```
If inputs are invalid (that means the blackboard identifier is unknown or the
   time-out value is “out of range”) Then
   Return INVALID_PARAM (as return code);
Else If a message is currently displayed on the specified blackboard Then
        send this message and return NO_ERROR;
Else If the time-out value is zero Then
        Return NOT_AVAILABLE;
Else If preemption is disabled or the current process is the error handler Then
        Return INVALID_MODE;
Else
       set the process state to waiting;
       If the time-out value is not infinite Then
          initiate a time counter with duration time-out;
       EndIf;
       ask for process scheduling (the process is blocked and will return to
       “ready” state by a display service request on that blackboard from
       another process or time-out expiration);
       If expiration of time-out Then
          Return TIMED_OUT;
       Else
          the output message is the latest available message of the blackboard;
          Return NO_ERROR;
       EndIf;
EndIf
```

Figure 6: Informal specification of the read_blackboard service.

Modeling of APEX services. To illustrate the approach adopted here, let us consider a typical APEX service: the *read_blackboard* service. It enables messages to be displayed and read in a blackboard. Input parameters are the blackboard *identifier* and a *time-out* duration (that limits the waiting time on a request, when the blackboard is empty). Output parameters are a *message* (defined by its address and size) and a *return code* (for the diagnostics of the service request). A typical informal specification of the service [Air97b] is given on FIG. 6.

We first define an abstract formal description corresponding to the service (see FIG. 7). This description expresses a certain number of properties. For example, `(s.2)` specifies logical instants at which a return code is produced. The variable `C_return_code` is a local boolean signal that carries the value *true*[4] whenever a return code is received on a read request (in other words, `C_return_code` represents the clock of the return code signal). For the moment, `C_return_code` appears in the description as a local signal. It will be defined during refinements of the abstract description. At this stage, we assume that there only exist signals such that properties in which it is involved are satisfied. Property `(s.1)` states that `C_return_code` and all input parameters are synchronous (i.e., whenever there is read request, `C_return_code` indicates whether or not a return code should be produced). Property `(s.3)` expresses the fact that messages are received on a read request only when the return code value is `NO_ERROR`.

Lines `(d.1)` and `(d.2)` give dependency relations between input and output parameters. In SIGNAL, the notation `x -> y` expresses a dependency relation between two signals `x` et `y` within a logical instant (`y` is also said to be preceded by `x`). For instance, `(d.2)` states that `message` and `length` are preceded by `timeout` and `board_ID`, at the logical instants where the return code carries the value `NO_ERROR`.

The level of detail provided by a description like the one given in FIG. 7 is expressive enough to check, for instance, the conformance of a component model during its integration in a system described in the same formalism. Here, the description exhibits the interface properties of the *read_blackboard* service. In particular, it gives conditions that describe when a message is received by a process on a read request. However, the description does not specify exactly **how** messages are obtained.

The specifications given in [Air97b] sometimes are imprecise. As a result, this leads to ambiguities, which are not easily perceptible. Here, two possible implementations are distinguished for the *read_blackboard* service. They mainly depend on the interpretation of message retrieval. Let us consider a process P_1, which was previously blocked on a read request in a blackboard, and now released on a display request by another process P_2:

[4] On a *read_blackboard* service request, a return code retrieval is not systematic. For instance, when the *blackboard* is empty and the value of the input parameter `timeout` is not infinite (represented in [Air97b] by a special constant *INFINITE_TIME_VALUE*), the requesting process is suspended. In this case, *C_return_code* carries the value *false*. The suspended process must wait: either a message is displayed on the *blackboard*, or the expiration of the active time counter initialized with `timeout` (hence, the return code carries the value *TIMED_OUT*).

```
process READ_BLACKBOARD =
  { ProcessID_type process_ID; }
  ( ? Comm_ComponentID_type board_ID;
      SystemTime_type timeout;
    ! MessageArea_type message;
      MessageSize_type length;
      ReturnCode_type return_code; )
 (| (| { {board_ID, timeout} -> return_code } when C_return_code    (d.1)
     | { {board_ID, timeout} -> {message, length} }
            when (return_code = #NO_ERROR)                           (d.2)
     |)
  | (| board_ID ^= timeout ^= C_return_code                          (s.1)
     | return_code ^= when C_return_code                             (s.2)
     | message ^= length ^= when (return_code = #NO_ERROR)           (s.3)
     |)
  |) where boolean C_return_code
```

Figure 7: Abstract description of the *read_ blackboard* service .

1. some implementations assume that the message read by P_1 (the suspended process) is the same as the one just displayed on the blackboard P_2;

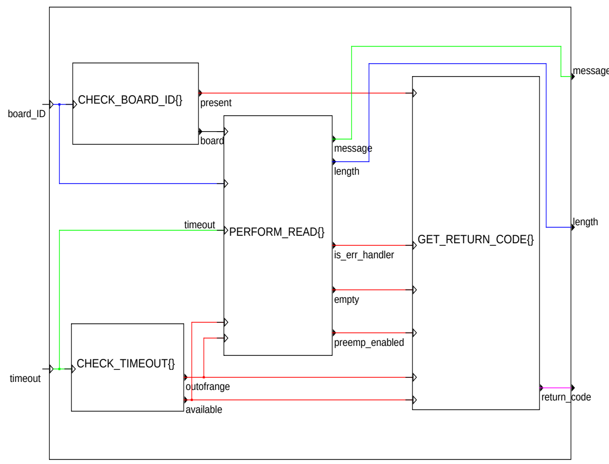
2. there are other implementations that display the message retrieved by P_1 *when the execution of* P_1 *gets resumed* (as a matter of fact, a higher priority process could be ready to execute when P_1 gets released). So, P_1 will not necessarily read the message displayed by P_2 since the message may have been overwritten when its execution is resumed.

As one can notice, the level of detail of the model described in FIG. 7, although very abstract, allows to cover both interpretations of the *read_ blackboard* service. In practice, we observe that these interpretations can be useful depending on the context.

- Implementations of type (1) may be interesting when all the messages displayed on the blackboard are relevant to the process P_1. Every message must be retrieved. However, even if using a blackboard for such message exchanges appears cheaper than using a buffer (in terms of memory space required for message queuing, and of blocked processes management), it would be more judicious to consider a buffer for such communications since there is no loss of messages.

- On the other hand, implementations of type (2) find their utility when P_1 does not need to retrieve all displayed messages. For instance, P_1 only needs to read *refreshed data* of the same type. In that case, only latest occurrences of messages are relevant.

The presence of ambiguities as illustrated above justifies a model refinement design approach, where abstract descriptions enable general descriptions that can be refined progressively in order to derive a particular implementation. Here, the way messages are retrieved during the *read_ blackboard* service call is not clear. The model given in FIG. 7 allows to describe such a situation.




```
(| board_ID ^= timeout ^= present ^= outofrange ^= available
      ^= C_return_code
 | board ^= empty ^= when present
 | message ^= length ^= when (not empty)
 | is_err_handler ^= when empty when available
 | preemp_enabled ^= when (not is_err_handler)
 | C_return_code := (when ((not present) or outofrange)) default
                    (when empty when (not available)) default
                    (when ((not preemp_enabled) default is_err_handler)) default
                    (when (not empty)) default
                    false
 | return_code ^= when C_return_code
 |)
```

Figure 8: Refined description of the *read_ blackboard* service and clock relations between signals.

A more detailed version of the service description is illustrated in FIG. 8. It relies on the second interpretation. In this version, four main sub-parts are distinguished. Sub-parts `CHECK_BOARD_ID` and `CHECK_TIMEOUT` verify the validity of input parameters `board_ID` and `timeout`. If these inputs are valid, `PERFORM_READ` tries to read the specified blackboard. Afterward, it sends the latest message displayed on the blackboard (its area and size are specified by `message` and `length`). It also transmits all the necessary informations to `GET_RETURN_CODE`, which defines the final diagnostic message of the service request.

For example, when signals `empty` and `preemp_enabled` respectively carry the values `true` and `false`, the sub-part `GET_RETURN_CODE` sends `INVALID_MODE` as `return_code` (that means the service caller is suspended until a message becomes available, and no other process can execute during the suspension because preemption is not enabled in the current operating mode). In the case of invalid inputs (e.g. `board_ID` is an unknown identifier within the partition, or `timeout` is "out of range"), informations are still sent to `GET_RETURN_CODE` by `CHECK_BOARD_ID` and `CHECK_TIMEOUT` in order to determine the value of the return code. A complete description of the service can be found in [GG02].

The other APEX services are modeled in a similar way as the *read_ blackboard* service. These services can be used to describe process management, communication and synchronization between processes, etc. The next section presents the modeling of the *partition-level OS*, which is in charge of controlling the execution of processes within a partition.

Modeling of the partition-level OS. The role of the partition level OS is to ensure the correct concurrent execution of processes within the partition (each process must have exclusive control on the processor). A sample model of the partition level OS is depicted in FIG. 9.

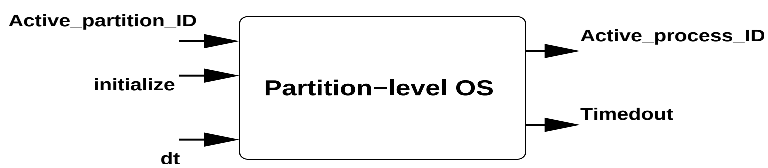


Figure 9: Interface of the partition level OS model.

The notions taken into account for the modeling of the *partition level OS* are mainly: *process management* (e.g. create, suspend a process), *scheduling* (including the definition of process descriptors and a scheduler), *time management* (e.g. update time counters), *communications*, and *synchronizations* between processes. The APEX interface provides a major part of required services to achieve the notions mentioned above. However, in order to have a complete description of the *partition level OS* functionalities, we added additional services to our library. These services allow us to describe process scheduling within a partition and they also allow to update time counters. Their description can be found in [GG03]. A case study using these services is presented in [GGL04]. Here, we only present the generic interface of the *partition level OS* (cf. FIG. 9). We explain how it interacts with the other subparts of its containing partition, in particular processes.

In FIG. 9, the input `Active_partition_ID` represents the identifier of the running partition selected by the module-level OS, and it denotes an execution order when it identifies the current partition (the activation of each partition depends on this signal. It is produced by the module-level OS, which is in charge of the management of partitions in a module). The presence of the input signal `initialize`, which corresponds to the initialization phase of the partition: creation of all the mechanisms and processes contained in the partition. Whenever the partition executes, the `PARTITION_LEVEL_OS` selects an active process within the partition. The process is identified by the value carried by the output signal `Active_process_ID`, which is sent to each process. The signal `dt` denotes duration information corresponding to process execution (more precisely, the duration of the current "block" of actions executed by an active process - we will come back on this point in the paragraph that presents the modeling of IMA processes). It is used to update time counter values. The signal `timedout` produced by the partition-level OS carries information about the current status of the time counters used within the partition. For instance, a time counter is used for a wait when a process gets interrupted on a service request with time-out. As the partition-level OS is responsible for the management of time counters, it notifies each interrupted process of the partition with the expiration of its associated time counter. This is reflected by the signal `timedout`.

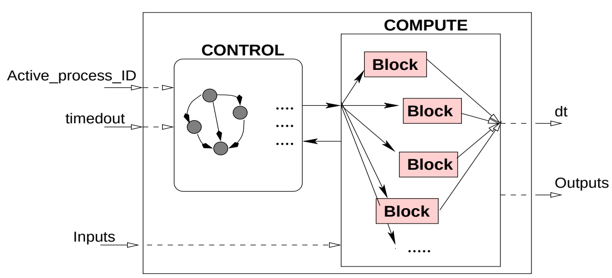


Figure 10: ARINC process model.

Modeling of IMA processes. The definition of an ARINC process model basically takes into account the computation and control parts of an ARINC process. This is depicted in FIG. 10. Two sub-components are clearly distinguished within the model: *CONTROL* and *COMPUTE*. Any ARINC process is seen as a reactive component, that reacts whenever an execution order (denoted by the input `Active_process_ID`) is received. The input `timedout` notifies processes of time-out expiration, while the output `End_Processing` is emitted by the process after completion. In addition, there are other inputs (respectively outputs) needed for (respectively produced by) the process computations. The CONTROL and COMPUTE sub-components cooperate to achieve the correct execution of the process model.

The CONTROL sub-component specifies the control part of the ARINC process. Basically, it is a transition system that indicates which statements should be executed when the process model reacts. It can be encoded easily by an automaton in SIGNAL. Whenever the input `Active_process_ID` (of numeric type) identifies the ARINC process, this process "executes". Depending on the current state of the transition system representing the execution flow of the process, a *block* of actions in the COMPUTE sub-component is selected to be executed *instantaneously* (this is represented by the arrow from CONTROL to COMPUTE in the figure).

The COMPUTE sub-component describes the actions computed by the process. It is composed of *blocks* of actions. They represent elementary pieces of code to be executed without interruption. The statements associated with a block are assumed to *complete within a bounded amount of time*. In the model, a block is executed instantaneously. Therefore, one must take care of what kinds of statements can be put together in a block. Two sorts of statements are distinguished. Those which may cause an interruption of the running process (e.g. a *read_ blackboard* request) are called *system calls* (in reference to the fact that they involve the partition level OS). The other statements are those that never interrupt a running process. Typically, data computation functions. They are referred to as *functions*.

For a correct execution, at most one system call can be associated with a block, and no other statement should follow this system call within the block. A block is executed instantaneously, but what would happen if the system call interrupts the running process? All the other statements within the block would be executed in spite of the interrupt, and this would not be correct. Furthermore when the process gets resumed, the whole block may not necessarily require to be re-executed. The process model proposed here is very simple. An execution of ARINC processes can be seen as a sequence of blocks, and preemption is represented by an occurrence of two consecutive blocks that belong to different processes in a sequence.

Global view of a partition model. FIG. 11 shows a rough view of a partition composed of three processes. In this model, the component `GLOBAL_OBJECTS` appears for structuring. In particular, communication and synchronization mechanisms used by processes (e.g. `buff`, `sema`) are created there.

The UML sequence diagram[5] depicted in FIG. 12 illustrates how the partition-level OS interacts with a process during the execution of the partition. After the initialization phase, the partition gets activated (i.e. when receiving *Active_ partition_ ID*). The partition-level OS selects an active process within the partition. Then, the CONTROL subpart of each process checks whether or not the concerned process can execute. In the diagram, this is denoted by the *optional* action (represented by a box labeled *opt*). In the case a process is designated by the OS, this action is performed: the process executes a block from its COMPUTE subpart, and the duration corresponding to the executed block is returned to the partition-level OS in order to update time counters. The execution of the model of

[5] UML Specification version 2.0: Superstructure – *Object Management Group* (`www.omg.org`).

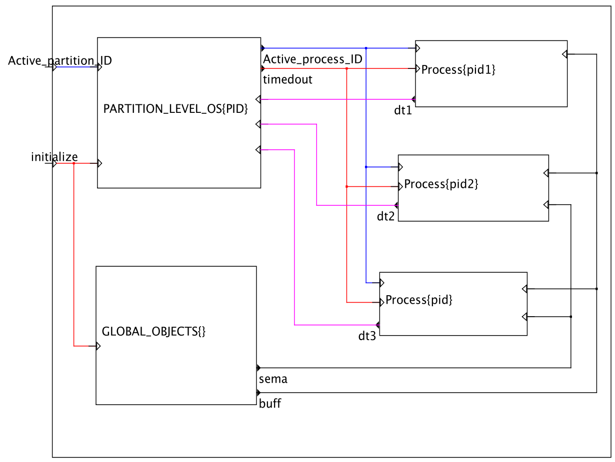


Figure 11: An example of partition model composed of three processes

the partition follows this basic pattern until the module-level OS selects a new partition to execute.

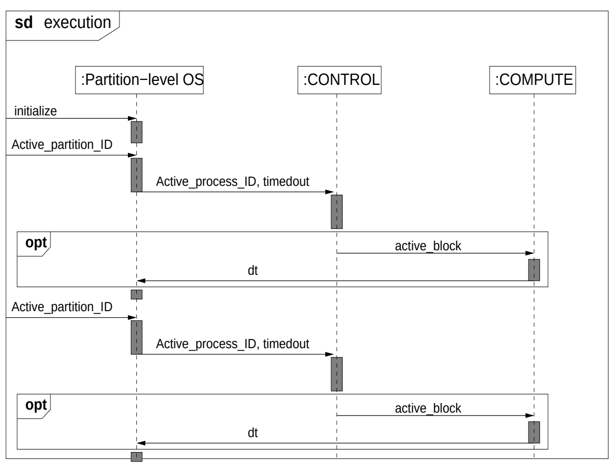


Figure 12: A sketch of the model execution.

4.2.3 Design by model refinement

By refinement, we mean a set of transformations allowing to define progressively, from an initial description P (a Signal program), further descriptions in the following way: at each step, a new description `Q` is obtained through the "instantiation" of intermediate variables by adding supplementary equations to `P`. Typically, this refinement process could modify non functional properties of `P` (e.g. temporal properties by introducing delays during the execution or by relaxing some synchronization relations), but its functional properties are strictly preserved.

The notions presented below have been introduced during the European project SACRES [GL99]. The goal was to define ways for generating distributed code from synchronous specifications (particularly SIGNAL programs). In the following, an application is represented by a SIGNAL program $P = P_1 \mid P_2 \mid ... \mid P_n$, where each sub-program P_i can be itself recursively composed of other sub-programs (i.e., $P_i = P_{i1} \mid P_{i2} \mid ... \mid P_{im}$). The following hypothesis are assumed:

1. considered programs P are initially *endochronous* [LTL03], hence deterministic (roughly speaking, an endochronous program can be executed separately);
2. they do not contain any circular definitions;
3. a set of processors $q = \{q_1, q_2, ..., q_m\}$; and
4. a function $locate : \{P_i\} \longrightarrow \mathcal{P}(q)$, which associates with each subpart of an application $P = P_1 \mid P_2 \mid ... \mid P_n$ a non empty set of processors (the allocation can be done either manually or automatically).

First transformation. Let us consider a SIGNAL program $P = P_1 \mid P_2$, as illustrated in FIG. 13. Each sub-program P_i (represented by a circle) is itself composed of four sub-programs P_{i1}, P_{i2}, P_{i3} and P_{i4}. The program P is distributed on two processors q_1 and q_2 as follows:

$$\begin{array}{ll} \forall i \in \{1,2\}\ \forall k \in \{1,2\}, & locate(P_{ik}) = \{q_1\}, \quad and \\ \forall i \in \{1,2\}\ \forall k \in \{3,4\}, & locate(P_{ik}) = \{q_2\} \end{array}$$

Hence, P can be rewritten into $P = Q_1 \mid Q_2$, where

$$\begin{array}{rl} Q_1 = & P_{11} \mid P_{12} \mid P_{21} \mid P_{22}, \quad and \\ Q_2 = & P_{13} \mid P_{14} \mid P_{23} \mid P_{24} \end{array}$$

The sub-programs Q_1 and Q_2 resulting from the partitioning of P are called ***s-tasks*** [GL99]. This transformation yields a new form of the program P that reflects a multi-processor architecture. It also preserves the semantics of the transformed program (since it simply consists in program rewriting).

Second transformation. We want to refine the level of granularity resulting from the above transformation. For that, let us consider descriptions at processor level (in other

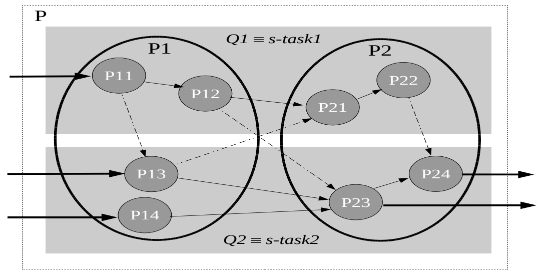


Figure 13: Decomposition of a Signal process into two s-tasks Q_1 and Q_2.

words, s-tasks). We are now interested in how to decompose s-tasks into fine grain entities. An s-task can be seen as a set of *nodes* (e.g. P_{11}, P_{12}, P_{21} *and* P_{22} in Q_1). In order to have an optimized execution at the s-task level, nodes are gathered in such a way that they can be executed atomically. So, we distinguish two possible ways to define such subsets of nodes, also referred to as *clusters* : either they are composed of a single Signal primitive construct, or they contain more than one primitive construct. The former yields a finer granularity than the latter. However, from the execution point of view, the latter is more efficient since more actions can be achieved at a same time (i.e. atomically).

The definition of atomic nodes use the following criterion: all the expressions present in such a node depend on the same set of inputs. This relies on a *sensitivity analysis* of programs. We say that a causality path exists between a node N_1 (resp. an input i) and a node N_2 if there is at least one situation where the execution of N_2 depends on the execution of N_1 (resp. on the occurrence of i). In that case, all the possible intermediate nodes are also executed.

Definition 1 *Two nodes N_1 and N_2 are sensitively equivalent iff for each input i: there is a causality path from i to $N_1 \Leftrightarrow$ there is a causality path from i to N_2.*

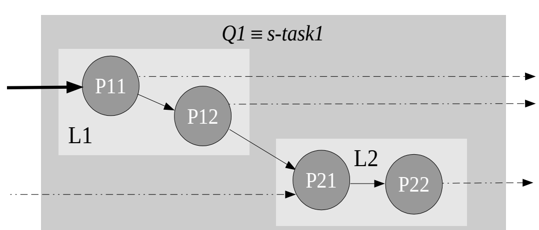


Figure 14: Decomposition of an s-task into two clusters L_1 and L_2.

Sensitively equivalent nodes belong to the same cluster. Inputs always precede outputs within a cluster. Also, if a transformed program is initially endochronous, the resulting clusters are also endochronous (this ensures a deterministic execution of each cluster). FIG. 14 shows a decomposition of the s-task Q_1 into two clusters $L1$ and L_2. The input of the sub-program $P11$ (bold-faced arrow) is originally an input of P. The other arrows represent communications between s-tasks (These message exchanges are local to P). We can notice that after this second transformation, the semantic equivalence of the initial program and the resulting one is strictly preserved.

The two transformations presented above describe a partitioning of SIGNAL programs following a multi-task multi-processor architecture. The instantiation of such a description in the IMA model consists in using the ARINC component models we have introduced in Section 4.2.2 (APEX services, processes, partitions).

Instantiation of SIGNAL programs in the IMA model. We first present this instantiation at processor level then the approach could be generalized to the multi-processor level. From the above transformations, a processor can be considered as a graph where nodes are represented by clusters. Therefore, the partitioning of a given SIGNAL program following the IMA architecture model is obtained through the following steps :

- **Step 0: Distribute the program on the available processors.** Here, we assume a given distribution function. The program is transformed into s-tasks. In practice, this step is often an expert matter. However, there exist tools that could strongly help to achieve this kind of task (e.g SYNDEX [GS03]).

- **Step 1: For each processor, transform the associated s-task into a graph of clusters.** This task is done automatically by the SIGNAL compiler.

- **Step 2: For each processor, associate clusters with partitions/processes.** The first decision about the graph of clusters resulting from the previous step consists in choosing a partitioning of clusters into IMA partitions/processes. In other words, we have to identify clusters that can be executed within the same partition/process. In our simple example, we decide to model the graph associated with $Q1$ (cf. FIG. 14) by one partition. Once partitions are chosen, the graph corresponding to each of them is decomposed into sub-graphs. These contain the clusters that should be executed by the same process. In the example, clusters associated with the "partition $Q1$" form the set of instruction blocks of a single process. The decomposition of a graph of clusters into partitions and processes must be done with respect to some coherent criterion. For instance, it can be very interesting to put clusters that strongly depend on each other together in a same partition/process. This would greatly reduce inter-process communication costs. In the next step, the program can be effectively instantiated using our building blocks.

$$
\begin{array}{rlll}
 & & p-OS & \text{(partition-level OS)} \\
 & & cont_i & \text{(``control`` part of a process } p_i\text{)} \\
 & & b_{ij} & \text{(block)} \\
p & +::= & p-OS \mid p_1 \mid \ \ldots \ \mid p_n & \text{(partition)} \\
p_i & ::= & cont_i \mid comp_i & \text{(process)} \\
comp_i & ::= & b_{i1} \mid \ \ldots \ \mid b_{im_i} & \text{(``compute'' part of a process } p_i\text{)}
\end{array}
$$

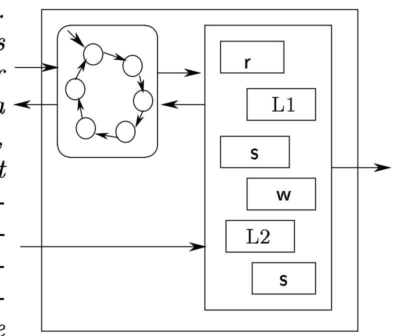
Figure 15: Modeling rules of IMA partitions.

- **Step 3: Instantiate the program in the IMA model.** Two phases are considered: first, we instantiate processes then partitions. An overview of the basic components used is given in FIG. 15. The symbol "|" denotes the synchronous composition. The following transformations are defined:

  1. Description of the process associated with a set of clusters:
     - The definition of the CONTROL part of the process relies on dependencies between clusters. Clusters are executed sequentially with respect to these dependencies.
     - Each cluster is "embedded" in a block within the COMPUTE part of the process.
     - The internal communications between the clusters of a sub-graph associated with a process are modeled using local state variables (i.e. those defined by the *delay* primitive construct). These variables enable to memorize exchanged data. On the other hand, communications between sub-graphs of clusters from different processes are modeled with APEX services. For each entry point (resp. exit point) of a sub-graph, a block containing a suitable communication or synchronization service call is added in the COMPUTE part of the associated process model. When the process becomes active, this block is executed just before (resp. after) the block that contains the cluster concerned by the entry point (resp. the exit point). The choice of the suitable service to call here depends on the desired type of communication. For instance, if one needs to use a bounded message queue, services associated with a buffer are preferred to those related to a blackboard, which are more appropriate for message exchanges via one memory-place. The services associated with semaphores are used for synchronization.
  2. Description of the partition associated with a set of clusters:
     - The component corresponding to the partition-level OS (containing among other things the scheduler, which manages process execution within the partition) is added to the processes defined at the previous phase.
     - The communication and synchronization mechanisms used by the APEX services added in the previous phase are created (for instance, for a *send_buffer*

service call, a *buffer* should be created in which the messages can be stocked). This creation is done for example within the `GLOBAL_OBJECTS` sub-part of the partition, as illustrated on FIG. 11.

Example *On the right, we outline a process model resulting from the transformation of* $Q1$. *There are six blocks where two contains clusters* $L1$ *and* $L2$. *The other blocks have been added for communication: r, s and w respectively denote a read request (receive_ buffer or read_ blackboard), an event notification (set_ event), and an event notification waiting (wait_ event). The automaton described in the control part gives the execution order of the blocks with respect to the precedence constraints of the cluster graph. The corresponding partition is obtained by considering the phase 2 of **Step 3**, in the current section.*



On each processor with multiple partitions, a model of a partition scheduler is required. Partition management is based on a time sharing strategy. Therefore, we have to compose a component (corresponding to the module-level OS - see Section 4.2.1) with partition models. A model of such a component is similar to the partition-level OS in that its definition relies on the use of APEX services, except that the scheduling policy differs.

In Section 3.2, we mention that a great advantage of SIGNAL-based modeling is the possibility to formally analyze descriptions. In particular, timing issues can be addressed using the performance evaluation technique implemented in POLYCHRONY.

4.2.4 A modular approach for timing analysis

The temporal analysis of the application SATMAINT exposed in this section is based on a technique presented in [KL96]. It relies on the principle introduced in Section 3.2, which consists of using an observer program to check properties of a given program.

Overview of the technique. As a general observation, a SIGNAL program is recursively composed of sub-processes, where elementary sub-processes are primitive constructs, called *atomic nodes*. A transformation of such a program substitutes each of its signals x with a new signal representing availability dates $date\_x$, automatically replacing atomic nodes with their temporal model counter-part. The resulting time model is composed with the original functional description of the application (using the standard synchronous composition). Each signal x has the same clock as its associated date information $date\_x$. The simulation of the resulting program reflects both the functional and timing aspects of the original program. Obviously, a less strict temporal model can be designed in order to perform faster simulation (or formal verification). It is sufficient to consider more abstract representations either of the program or of its temporal model.

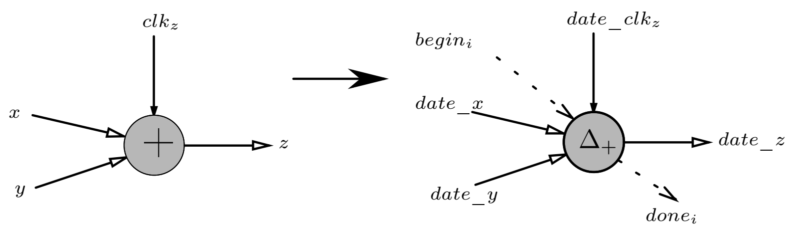


Figure 16: Node associated with `z := x + y` (left); and its temporal model (right).

The temporal interpretations of SIGNAL primitive constructs are collected in a *library of parameterized cost functions*. For a program to be interpreted, the library is extended with interpretations of external function calls and other separately compiled processes, which appear in the program. As an illustration, let us consider the following primitive construct: `z := x + y`. It is represented by the atomic node depicted by Figure 16, on the left hand side. Besides the input values `x` and `y`, this node also requires a clock information, denoted by the signal `clk_z`, which triggers the computation of the output value `z`. The associated temporal model is represented by the node illustrated on the right hand side. The SIGNAL program corresponding to this temporal model, called `T_Plus`, is depicted by Figure 17. In this model, `MAXn` denotes a SIGNAL process that returns the maximum value of `n` inputs among those that are present at a given instant (i.e., inputs are not constrained to be simultaneously present). The notations `type_x` and `type_y` represent the types of `x` and `y` respectively. The input signal `date_clk_z` is associated with the trigger clock `clk_z`. Signals `begin_i` and `done_i` have been added in order to express the end of execution of a given node so that the following nodes could be executed. The presence of `begin_i` in a node means that the preceding ones (following the scheduling order chosen for node execution), have already produced all their output dates. The presence of `done_i` means that the current node has calculated all its output dates (i.e., `done_i` becomes `begin_i+1`). The date of `z`, denoted by `date_z`, is the sum of the maximum date of inputs and the delay of the addition operation, some Δ_+. This quantity Δ_+ depends on the desired implementation, on a specific platform. It has to be provided in some way by the user, with respect to the considered platform. In the current implementation in POLYCHRONY, the value Δ_+ is provided by a function `DELTA_ADD` which has the types of the operands as parameters and which fetches the required value from some table. Following the same idea, each primitive construct of SIGNAL has been associated with its temporal interpretation. As a result of the compositionality of SIGNAL specifications, the same principle can be applied at any level of granularity.

In addition to the library of cost functions of primitive constructs, the implementation also requires *platform-dependent information* (e.g. the delay of the addition of two integer

```
process T_Plus{ type_x, type_y; }
 ( ? date_type date_x, date_y, date_clk_z, begin_i;
   ! date_type date_z, done_i; )
(| date_z := MAX2( MAX3( date_x, date_y, date_clk_z),
                   begin_i when ^date_z) + DELTA_ADD{type_x, type_y}()
 | done_i := (date_z default begin_i) cell ^done_i
 |);
```

Figure 17: A temporal interpretation of `z := x + y` in SIGNAL.

numbers on a given processor). For instance, the sum of integer numbers coded on 32 bits will take one cycle on a processor with a 32-bit adder (this is the case of the Intel Pentium IV processor). On the other hand, the same operation requires more than one cycle on a processor with only a 16-bit adder (like the Intel 8086 processor). In the example exposed in Figure 17, this information is obtained via the `DELTA_ADD` call.

Application to a real world case study. Let us consider an application (called SATMAINT) represented by a single partition model as it has been shown in [GGL04]. FIG. 18 shows a *co-simulation* of this application together with its temporal interpretation T_SATMAINT (the prefix notation "T_" stands for temporal).

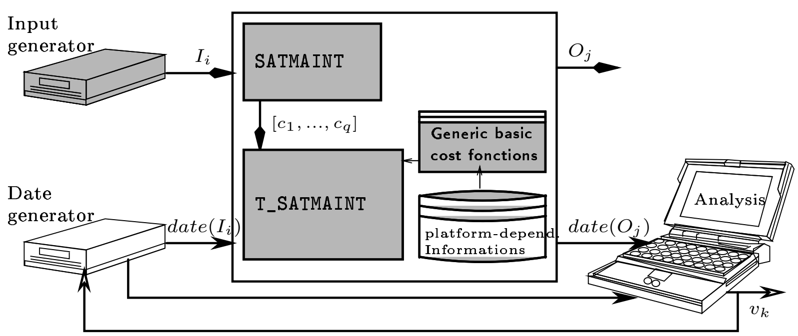


Figure 18: Co-simulation of SATMAINT with its temporal interpretation.

At each simulation step, the date of outputs $date(O_j)$ depends on the date of inputs $date(I_i)$ and the *control configuration* (represented by a "valuation" of boolean signals vector $[c_1, \ldots, c_q]$ computed in the original program). In fact, the control parts of SATMAINT and T_SATMAINT are identical, only data parts differ: in SATMAINT, the data part computes functional results while in T_SATMAINT, it yields date informations. The vector $[c_1, \ldots, c_q]$ contains intermediate boolean signals evaluated by the data part of SATMAINT that are needed by T_SATMAINT to compute output information. Note that in a straightforward approach, it is possible to provide a set of vectors that covers all the possible combinations for the control flow. A better way is to take into account the existing relationships between these booleans such as provided by the clock calculus of Signal (this is expressed through the composition of the original program and its temporal interpretation).

In practice, we mainly raise one difficulty about the implementation of the schema illustrated in Figure 18. It is due to the scalability issue which can become problematic when compiling large application programs. For example, the program resulting from the composition of the application model together with its temporal interpretation - represented by `(| SATMAINT | T_SATMAINT |)` - can be huge and may not facilitate the compiling. So, the adopted solution consists of using a modular evaluation approach, which is exposed in the sequel.

The modularity of the Signal language allows to construct a program from other programs by composition. Therefore, this principle also applies to the construction of the temporal interpretation of the partition SATMAINT, which is quite large. For that, we consider a splitting of the corresponding Signal model into subparts with reasonable size (e.g. such a subpart could be an IMA process). This will make them easier to address. So, for each subpart, we define an associated temporal interpretation. Afterwards, the resulting model can be composed with the concerned subpart. The program obtained from this composition is abstracted in order to take into account only information that is relevant for the considered observation (abstractions also contribute to reduce the size of a program). Finally, the global model that consists of the composition of the application with its temporal interpretation is obtained by composing abstracted subprograms.

Let $\mathtt{P} \equiv \mathtt{P}_1$ `| ... |` $\mathtt{P}_n$ denote the program corresponding to the application considered for temporal analysis. We want to define the program `P'` $\equiv$ `(| P | T_P |)` that will be used for the analysis, where `T_P` is the temporal model of `P`. The same program can be also rewritten as: `P'` $\equiv$ `P'`$_1$ `| ... |` `P'`$_n$. Each `P'`$_i$ denotes the composition of a subprogram $\mathtt{P}_i$ of `P`, with its associated temporal interpretation `T_P`$_i$. The following steps are identified in order to carry out experiments:

1. *Partial definition of temporal interpretations*: for each sub-program $\mathtt{P}_{i, i \in \{1,...,n\}}$ of `P`, we define the corresponding temporal model `T_P`$_i$.
2. *Composition of each subpart of the application with its associated temporal interpretation, then abstraction*: this step defines the subprograms `P'`$_i$ ($i \in \{1, ..., n\}$), which constitute the simulation program `P'` that we want to construct. The abstraction aims to keep only relevant information of these subprograms for the co-simulation. It follows that `P'`$_i \equiv \alpha(\mathtt{P}_i$ `|` `T_P`$_i)$, where α denotes the abstraction (e.g. a program can be abstracted by considering only its control part—boolean and synchronization

signals—or by approximating the value of numerical signals—for instance, by dealing with domains of intervals instead of point-wise domains).

3. *Construction of the global model for simulation*: this model results from the composition of subprograms $\texttt{P'}_i$, defined at the previous step, i.e.

$$\texttt{P'} \equiv \texttt{P'}_1 \texttt{ | ... | } \texttt{P'}_n.$$

On the other hand, we notice that the above method can be applied in a recursive way. Still considering the above program `P`, if the size of subprograms $\texttt{P}_i$ $(i \in \{1, ..., n\})$ is important, we can also use the same method for each one in order to define the corresponding $\texttt{P'}_i$ $(i \in \{1, ..., n\})$. The global simulation model then results from the composition of $\texttt{P'}_i$s.

To apply the method to the partition SATMAINT, we first decomposed it into subparts, represented by its processes (it contains eight processes). For instance, its process identified as `PROC_8` is composed of eight blocks. We begin by defining the interpretation of the COMPUTE subpart of this process:

$$\texttt{T\_COMPUTE} \equiv \texttt{T\_BLOCK\_0 | ... | T\_BLOCK\_7}.$$

In this composition, we suppose that the temporal model of each block is obtained without necessarily having to consider its decomposition as is done for subprograms of a larger size. The temporal interpretation of the CONTROL subpart of the process is determined in a similar way. The composition of both temporal models gives the model for `PROC_8`:

$$\texttt{T\_PROC\_8} \equiv \texttt{T\_COMPUTE\_8 | T\_CONTROL\_8}.$$

We proceed in the same way for the other subparts of the partition. Then, we compose the resulting temporal model with its associated program subpart. For `PROC_8`, it follows:

$$\texttt{(| PROC\_8 | T\_PROC\_8 |)}.$$

This process could be now simplified by considering approximations (or abstractions). This will facilitate the compiling of the global program. For instance, let us consider a subpart of the application that performs a complex operation, which requires a constant duration δ on a target platform (by "complex", we mean an operation that requires several elementary operations, e.g. product of two matrices). The temporal model of such a subpart can be defined in a simple way by adding the constant δ to the dates corresponding to inputs availability in order to compute dates associated with outputs. This interpretation is simpler than the one obtained by composing the temporal models of all intermediate operations that are carried out by the considered subprogram. Other simplifications consist of considering worst case execution times [PB00] in the definition of the temporal interpretation of a subprogram. Finally, interesting possible abstractions of subparts of the application can be obtained by considering only their control parts. They provide enough information for the co-simulation with the associated temporal models.

Observations. Modularity and abstraction play a central role for scalability in our design approach. Basically, the description of a large application is achieved by specifying first, either completely or partially (by using abstractions), sub-parts of the application. After that, the resulting programs can be composed in order to obtain new components. These components can be also composed and so on, until the application description is complete. The construction of the global simulation model of SATMAINT for temporal issues relies on the same principle as the description of the application itself. A crucial issue about the design of safety critical systems, such as avionics, is the correctness of these systems. In POLYCHRONY, the functional properties of a system can be checked using tools like the compiler or the model checker SIGALI. Here, we addressed temporal aspects of programs. For that, we used a technique consisting of co-simulating the program under analysis with an associated observer (also referred to as temporal interpretation) defined in SIGNAL. The observer is another program which has the same control as the observed one, but its data part reflects the temporal dimension of the analyzed program. Using SIGNAL for both the model of an application and its associated temporal interpretation results in unified descriptions upon which available tools and techniques remain applicable.

4.2.5 Some related work

We mention a few studies addressing the design of embedded real-time systems in the avionic domain. The first one is the COTRE approach [BRV+03]. Its main objective consists in providing the designer with a methodology, an Architecture Description Language (ADL) called *Cotre*, and an environment to describe, verify and implement embedded avionic systems. The Cotre language distinguishes two different views for descriptions: a user view expressed using the *Cotre for User* language (termed *U-Cotre*) and a view for verification (termed *V-Cotre*). In fact, the latter plays the role of an intermediate language between U-Cotre and certain existing verification formalisms (e.g. timed automata, timed Petri nets). The authors argue that the use of formal techniques is one of the main differences between the Cotre language and other ADLS. COTRE is closely related to the approach based on the *Avionics Architecture Description Language* (AADL), which is developed by the International Society of Automotive Engineers (SAE) [AAD02]. It is dedicated to the design of the software and hardware components of an avionic system and the interfaces between those components. The AADL definition is based on METAH (an ADL developed by Honeywell) [Ves97]. It permits the description of the structure of an embedded system as an assembly of software and hardware components in a similar way. The AADL draft standard also includes a UML profile of the AADL. This enables the access to formal analysis and code generation tools through UML graphical specifications.

While these approaches combine various formalisms and tools for the design of embedded real-time systems, our approach relies on the single semantic model of the SIGNAL language.

It is very important to have a common framework in order to guarantee the correctness of the designs. Modularity allows to overcome scalability problems.

Among specific studies related to IMA, we mention those concerning the two-level hierarchical scheduling aspects within IMA systems. In [AW96], Audsley and Wellings introduced a scheduling approach for APEX applications. In [LKY$^+$00], Lee et *al.* presented algorithms in order to produce cyclic partition and channel schedules for IMA-based avionic systems. The technique we illustrated in this paper for temporal analysis provides information on execution times of partitions. Thus, this information could be used when taking decisions in processor allocation to partitions. Further expected benefits of defining our approach in a formal framework are the available techniques and tools that help to address some critical issues of IMA such as the partitioning, which still need to be further explored by researchers. Indeed, in current industrial practices, avionic functions with high critical level are designed using federated architectures (for instance, this is the case for the future Airbus A380). This is likely due to the fact that partitioning raises several questions that are not sufficiently addressed yet. Among these questions, we can mention the correctness of a partitioning, which is crucial. A formal description of partitioning requirements is proposed by Di Vito [Di 99], using the language of PVS (Prototype Verification System). However, this description only concerns space partitioning (time partitioning is not addressed). The use of a data-flow representation such as in SIGNAL can allow to define a correct-by-construction partitioning, based on a so-called *sensitivity analysis* [Sac97]. Being able to guarantee the correctness of a given partitioning can help to reduce IMA certification efforts. A study addressing this last issue has been done by Conmy and McDermid [CM01], who propose a high level failure analysis of IMA. The analysis is part of an overall IMA certification strategy. Finally, a presentation of the IMA-based communication network designed for the future Airbus A380 is given by Sánchez-Puebla and Carretero in [SPC03].

In the next section, we present a short study that also illustrates the general methodology introduced in section 4. We present a technique to import a resource constrained, multi-threaded, Real-Time Java (RTJ) program, together with its runtime system API, into POLYCHRONY [TGB$^+$03]. We put this modeling technique to work by considering a formal, refinement-based, design methodology that allows for a correct by construction remapping of the initial threading architecture of a given Java program on either a single-threaded target or a distributed architecture. This technique enables the generation of stand-alone (JVM-less) executables and remapping of threads onto a given distributed architecture or a prescribed target real-time operating system. As a result, it allows for a complete separation between the virtual threading architecture of the functional-level system design (in Java) and its actual, real-time and resource constrained implementation.

4.3 Re-engineering of Real-Time Java programs within POLYCHRONY

We consider embedded software implemented by resource constrained[6] multi-threaded programs on a specific runtime sub-system, represented here by the *real-time java virtual machine*, which we call its execution architecture. On the other hand, we restrict ourselves to a subset of the *Ravenscar* High Integrity Profile (HIP) specification [KWK02] for RTJ.

. . . / . . .
class SchedulingParameters
class AsyncEvent
  class SporadicEvent
  class SporadicInterrupt
class AsyncEventHandler
    (implements Schedulable)
  class BoundAsyncEventHandler
	class RealtimeThread
	    (implements Schedulable)
	  class Initializer
	  class NoHeapRealtimeThread
	      (implements Schedulable)
	    class PeriodicThread
	class SchedulingParameters
	. . . / . . .

Table 1: Excerpt of the considered RTJ package class hierarchy.

4.3.1 A Real-Time Java Profile

The profile [KWK02] consists of a subset of the RTJ specification [BBD+01] aimed at meeting non-functional requirements of airborne systems. Table 1 details the most significant features of this profile (however, we do not consider memory management). Extensions to thread management classes are provided to allow for the simulation of real-time threads and event handlers. The class AsyncEvent and its sub-classes define data-structure to encapsulate hardware interrupts and events to be recycled within the real-time JVM by appropriate handlers (class AsyncEventHandler and sub-classes) according to the pace of the real-time kernel. Processing in nominal mode in a real-time application is performed using real-time and periodic threads (classes RealTimeThread and PeriodicThread), which allow to schedule execution of a sequential code according to real-time constraints (period, deadline, duration, priority) set using shared data-structures defined in the SchedulingParameters class.

The HIP profile also describes programming guidelines in order to meet structural and functional requirements imposed by certification authorities. The syntactic simplicity of these requirements make them at the same time easy to *i)* translate into programming guidelines by users, *ii)* implement as syntactic checks with the help of a modeling tool, and *iii)* analyze using rate monotonic analysis techniques [KRP+93]:

[6] It is common sense to restrict ourselves to programs where all objects are first created and initialized to elaborate the application architecture. Then, threads implement reactions to inputs in the nominal phase of execution and do not allocate any new object (to comply with certification requirements in software design).

- a program consists of a fixed number of threads.
- thread and memory allocation is performed during service startup.
- no dynamic memory management during operational service.
- threads have access to the scope of the program.
- threads are either periodic or sporadic.
- threads use synchronization to avoid priority inversion.

4.3.2 A Real-Time Java plugin for POLYCHRONY

Following our methodology (see Section 4), a RTJ program representing an application is decomposed by distinguishing the *application functionalities*, the *application architecture* and *execution architecture*.

In particular, we need to identify the architecture of the application and its periodic threads and event handlers. This architecture (i.e. threads that constitute the application and their parameters, and shared data-structures) can be obtained by scanning the main (initialization) class of the Java program as well as the init methods of thread classes. The periodic threads and event handlers are described in the run methods of the thread classes. The init methods make use of operating system support (the RTJ API), which is modeled by the APEX-ARINC services of POLYCHRONY (see Section 4.2.2).

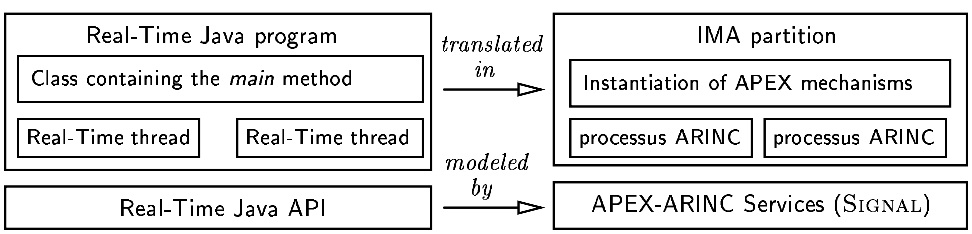


Figure 19: From Real-Time Java programs to SIGNAL models.

FIG. 19 carries out the main idea of the approach. Its foundation is provided by the APEX services, which model the RTJ API and correspond to the virtual machine. The structure of the actual Java program, which describes the functional threading is translated into instances of APEX services and IMA partitions (threads and event handlers are translated into SIGNAL models of IMA processes).

The approach is illustrated on a simple example program representing an Even-Parity Checker (EPC). This program consists of three threads (FIG. 20, left-hand side): an input/output interface thread, a master even parity-checking thread and a slave ones bit-counting thread. The Io thread sends a start signal to the Even thread and waits for the signal done to read the result. On the receipt of start, Even reads the input data, sends it to the thread Ones and notifies it with the istart signal. Ones counts the number of bits of the input data that are true and notifies Even of the completion. Even then checks if the result is even or not and notifies Io of the end of checking. The corresponding SIGNAL model is given on the right-hand side, in FIG. 20. It is represented by a single IMA partition composed of three processes associated with each thread.

```
import javax.realtime.*;
class EPC {
  public static int inport, outport, data,
   ocount;
  public static boolean start, done, istart,
   idone;
  public static void main (String argv[]){
   IO Io = new IO();
   EVEN Even = new EVEN();
   ONES Ones = new ONES();
   Io.start();
   Even.start();
   Ones.start();
  }
}
```

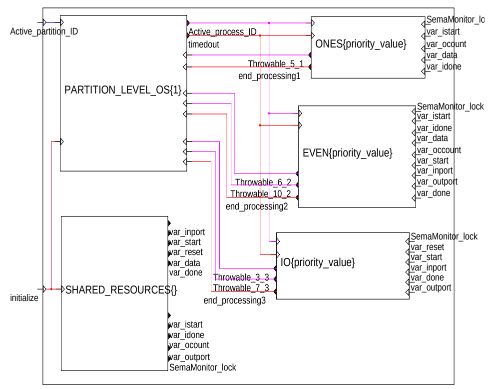


Figure 20: A model of a distributed implementation of an even-parity checker.

To describe actions achieved by each RTJ thread, we consider the JIMPLE intermediate format [VRH98]. It consists of explicitly typed, stack-less, 3-address statements that manipulate constants and references. Its accompanying SOOT[7] toolbox provides an appropriate front-end to resolve high-level object-oriented features and perform specific optimizations. This allows us to focus on the translation of the JIMPLE imperative notation into the polychronous data-flow design language SIGNAL [TGB$^+$03]. The resulting descriptions are then considered for transformations (e.g. the rethreading multi-threaded RTJ programs by global optimizations) or analysis. The POLYCHRONY workbench offers the required facilities to achieve that.

More generally, the technique presented in this section enables the integrated modeling, optimization, verification, and simulation of embedded systems in a functional subset of the Real-Time Java specification (compliant with certifiable software engineering requirements in avionics). It uses the multi-clocked synchronous design platform POLYCHRONY, which allows one to perform precise and aggressive optimizations and transformations that would be hard, yet impossible to achieve using common techniques.

[7] http://www.sable.mcgill.ca/soot.

5 Conclusions

In this paper, we argue that the polychronous framework favors reliable designs of embedded real-time systems. The central idea is the definition of a library of polychronous models of architecture components. As in METAH [Ves97] or VEST [SZP+03], this library includes active run-time components such as processes and functionalities of a real-time operating system.

We advocate a seamless design methodology including high level specifications using the modularity and re-usability features of the polychronous language SIGNAL, formal verification and performance evaluation, and automatic code generation. In such a context, the formal basis of SIGNAL is a key aspect for validation. This is essential to the design of safety critical systems.

Beyond the formal framework promoted by the present work for the design activity, it suggests a possible way to address the crucial issue of partitioning in integrated modular avionics architectures. Most modern aircraft still massively adopt the federated approach instead of IMA for highly critical functions. It seems that one main difficulty of that arises from the partitioning itself in such systems. We believe that the ideas exposed in this paper help to overcome this difficulty. For instance, the sensitivity analysis we use in our approach allows us to easily identify dependencies between sub-parts of a system, in a correct way. In this sense, the SYNDEX approach [GS03] can be also mentioned here. In SYNDEX, the partitioning of an application relies on the performance of the implementation platform. This can be combined with our sensitivity analysis within a new version of our methodology where the distribution function previously assumed at **Step 0** (see Section 4.2.3) is now replaced by a strategy based on quantitative criteria. We then observe a complementarity of SYNDEX and our approach.

From a scientific point of view, the main contribution of this paper concerns the modeling of (bounded) asynchronous mechanisms using the synchronous approach. There have been many studies on this topic [BS98] [Cas01] [HB02]. They turn out to promote *globally asynchronous locally synchronous* architectures (GALS) where a system is composed of subsystems/components that execute synchronously and communicate asynchronously. From a methodological point of view, various design approaches have been combined within a unique general methodology. We first considered a component-based approach in order to define a library of models. Then, we used these models to derive from a given SIGNAL description of an application, its corresponding IMA model by refining the initial description. We also showed how temporal issues or the resulting description can be addressed in a modular way. Moreover, we have illustrated a use of our library for a re-engineering of Real-Time Java programs within POLYCHRONY in order to analyze them. Finally, from an implementation point of view, this work led to the definition of synchronous models of APEX-ARINC 653 services. They are freely available together within the POLYCHRONY platform at `http://www.irisa.fr/espresso/Polychrony`. We faced the scalability question of the proposed approach by experimenting it on a real world application.

Acknowledgments. We wish to thank David Berner for his valuable comments on the previous version of this report.

References

[AAD02] AADL Coordination Commitee. Avionics architecture description language. In *AADL seminar*, Toulouse, France, Octobre 2002. Society of Automotive Engineers.

[ABL95] T.P. Amagbegnon, L. Besnard, and P. Le Guernic. Implementation of the dataflow language SIGNAL. In *Programming Languages Design and Implementation*, La Jolla, California, 1995. ACM Press.

[ADE+03] R. Alur, T. Dang, J.M. Esposito, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G.J. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. *IEEE Press*, 91(1):11–28, 2003.

[Air97a] Airlines Electronic Engineering Committee. ARINC report 651-1: Design guidance for integrated modular avionics. Technical Report 651, Aeronautical radio, Inc., Annapolis, Maryland, November 1997.

[Air97b] Airlines Electronic Engineering Committee. ARINC specification 653: Avionics application software standard interface. Technical Report 653, Aeronautical radio, Inc., Annapolis, Maryland, January 1997.

[AW96] N.C. Audsley and A.J. Wellings. Analysing APEX Applications. In *Proceedings of Real Time Systems Symposium (RTSS'96)*, Washington, DC, USA, 1996. IEEE Press.

[BBD+01] G. Bollela, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Harding, M. Turnbull, and R. Belliardi. The real-time specification for java. Technical report, The RealTime for Java expert group, December 2001.

[BCE+03] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[BLJ91] A. Benveniste, P. Le Guernic, and C. Jacquemot. Programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[BRV+03] Bernard Berthomieu, Pierre-Olivier Ribet, François Vernadat, J. L. Bernartt, Jean-Marie Farines, Jean-Paul Bodeveix, Mamoun Filali, Gérard Padiou, Pierre Michel, Patrick Farail, Pierre Gauffilet, Pierre Dissaux, and Jean-Luc Lambert. Towards the verification of real-time systems in avionics: the cotre approach. *Electronic Notes in Theoretical Computer Science*, 80:1–16, 2003.

[BS98] G. Berry and E. Sentovich. Embedding synchronous circuits in GALS-based systems. In *Proceedings of the Sophia-Antipolis conference on Micro-Electronics (SAME'98)*, Sophia-Antipolis, France, October 1998.

[Cas01] P. Caspi. Embedded control: From asynchrony to synchrony and back. In *Proceedings of the first International Workshop on Embedded Software (EMSOFT'01)*, Lake Tahoe, October 2001. Th. A. Henzinger and Ch. M. Kirsch, Eds., LNCS 2211, Springer Verlag.

[CM01] P. Conmy and J. McDermid. High Level Failure Analysis for Integrated Modular Avionics. In *Proceedings of the 6th Australian Workshop on Industrial Experience with Safety Critical Systems and Software*, Brisbane, Australia, June 2001.

[CPP+01] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys: a tool for the development and verification of real-time embedded systems. In *Proceedings of Computer Aided Verification, CAV'01*, Paris, France, July 2001. Lecture Notes in Computer Science 2102, Springer-Verlag.

[Di 99] B.L. Di Vito. A Model of Cooperative Noninterference for Integrated Modular Avionics. In *Proceedings of Dependable Computing for Critical Applications (DCCA-7)*, San Jose, CA, January 1999.

[DY95] C. Daws and S. Yovine. Two Examples of Verification of Multirate Timed Automata with KRONOS. In *Proceedings of the 16th IEEE Real Time Systems Symposium (RTSS'95)*, Pisa, Italy, December 1995. IEEE Press.

[GG02] A. Gamatié and T. Gautier. Synchronous modeling of modular avionics architectures using the SIGNAL language. Technical Report 4678, INRIA, December 2002. Available at `http://www.inria.fr/rrrt/rr-4678.html`.

[GG03] A. Gamatié and T. Gautier. Synchronous modeling of avionics applications using the SIGNAL language. In *Proceedings of the 9th IEEE Real-time/Embedded technology and Applications symposium (RTAS'03)*, Washington D.C., USA, May 2003. IEEE Press.

[GGL04] A. Gamatié, T. Gautier, and P. Le Guernic. An example of synchronous design of embedded real-time systems based on IMA. In *Proceedings of the 10th International Conference on Real-time and Embedded Computing Systems and Applications (RTCSA'04)*, Gothenburg, Sweden, August 2004. LNCS, Springer Verlag.

[GL99] T. Gautier and P. Le Guernic. Code generation in the SACRES project. In *Safety-critical Systems Symposium, SSS'99, Springer.*, Huntingdon, UK, February 1999.

[GMGW01] D. Goshen-Meskin, V. Gafni, and M. Winokur. SAFEAIR: An integrated development environment and methodology. In *Proceedings of the INCOSE 2001*, Melbourne, July 2001.

[GS03] T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *Formal Methods and Models for Codesign Conference*, Mont Saint-Michel, France, June 2003.

[HB02] N. Halbwachs and S. Baghdadi. Synchronous modelling of asynchronous systems. In *Conference on Embedded Software (EMSOFT'02)*, pages 240–251, Grenoble, France, October 2002. J. Sifakis and A. Sangiovanni-Vincentelli, Eds, LNCS 2491, Springer Verlag.

[HHK01] T.A. Henzinger, B. Horowitz, and Ch.M. Kirsch. Embedded control systems development with giotto. In *Proceedings of LCTES*. ACM SIGPLAN Notices, 2001.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology*, pages 83–96, Enschede, The Netherlands, 1993. Springer Verlag 1994, ISBN 3-540-19852-0.

[KL96] A. Kountouris and P. Le Guernic. Profiling of SIGNAL programs and its application in the timing evaluation of design implementations. In *Proceedings of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems*, pages 6/1–6/9, Bristol, UK, February 1996. HP Labs.

[KRP+93] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, Boston, MA, 1993.

[KWK02] J. Kwon, A.J. Wellings, and S. King. A high integrity profile for real-time java. In *Joint ACM Java Grande Conference*. ACM press, 2002.

[Lee00] E.A. Lee. What's ahead for embedded software? *IEEE Computer Magazine*, 33(9):18–26, September 2000.

[Lee01] E.A. Lee. Overview of the Ptolemy project. Technical Report UBC/ERL M01/11, University of California, Berkeley, March 2001.

[LKY+00] Y.-H. Lee, D. Kim, M. Younis, J. Zhou, and J. McElroy. Resource Scheduling in Dependable Integrated Modular Avionics. In *Proceedings of the International Conference on Dependable Systems and Networks*, April 2000.

[LTL03] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for System Design. *Journal for Circuits, Systems and Computers*, 12(3):261–304, April 2003.

[MBLL00] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the SIGNAL environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4):325–346, October 2000.

[PB00] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

[Pnu02] A. Pnueli. Embedded Systems: Challenges in Specification and Verification. In *Proceedings of the second International Conference on Embedded Software (EMSOFT 2002)*, pages 252–265, Grenoble, France, October 2002. J. Sifakis and A. Sangiovanni-Vincentelli, Eds, LNCS 2491, Springer Verlag.

[Rom02] J. Romberg. Model-based deployment with autofocus: a first cut. In *14th Euromicro Conference on Real Time Systems (ECRTS'02), Work In Progress session*, pages 41–44, Vienna, Austria, June 2002. IEEE Press.

[Rus99] J. Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical Report CR-1999-209347, NASA Langley Research Center, June 1999. Available at `http://www.csl.sri.com/users/rushby/partitioning.html`.

[Sac97] Sacres Consortium. The semantic foundations of SACRES. Technical Report EP 20897, March 1997.

[Sif01] J. Sifakis. Modeling real-time systems - challenges and work directions. In *Proceedings of the first International Workshop on Embedded Software (EMSOFT 2001)*, Tahoe City, October 2001. LNCS 2211, Springer Verlag.

[SPC03] M.A. Sánchez-Puebla and J. Carretero. A new Approach for Distributed Computing in Avionics Systems. In *Proceedings of the 1st International Symposium on Information and Communication Technologies*, Dublin, Ireland, 2003.

[SZP+03] J.A. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis. VEST: An aspect-based composition tool for real-time systems. In *Proceedings of the 9th IEEE Real-time/Embedded technology and Applications symposium (RTAS'03)*, Washington D.C., USA, May 2003. IEEE Press.

[TBS+04] J.-P. Talpin, D. Berner, S.K. Shukla, P. Le Guernic, A. Gamatié, and R. Gupta. A behavioral type inference system for compositional system design. In *Proceedings of the International Conference on Application of Concurrency to System Design.*, pages 47–56, Hamilton, Ontario, June 2004. IEEE Press.

[TGB+03] J.-P. Talpin, A. Gamatié, D. Berner, B. Le Dez, and P. Le Guernic. Hard real-time implementation of embedded systems in java. In *International Workshop on Scientific Engineering of Distributed JAVA Applications*, Luxembourg, November 2003. Lectures Notes in Computer Science, Springer Verlag.

[TLS+04] J.-P. Talpin, P. Le Guernic, S.K. Shukla, R. Gupta, and F. Doucet. Formal refinement checking in a system-level design methodology. *Fundamenta Informaticae, IOS Press*, August 2004.

[Ves97] S. Vestal. MetaH support for real-time multi-processor avionics. In *IEEE Workshop on Parallel and Distributed Real-Time Systems*, Geneva, Switzerland, April 1997. IEEE Press.

[VRH98] R. Vallee-Rai and L.J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. Technical Report 1998-4, McGill University, July 1998. Available at `http://www.sable.mcgill.ca/publications/techreports`.

[Wir01] N. Wirth. Embedded systems and real-time programming. In *Proceedings of the first International Workshop on Embedded Software (EMSOFT 2001)*, pages 486–492, Tahoe City, California, October 2001. Th. A. Henzinger and Ch. M. Kirsch, Eds., LNCS 2211, Springer Verlag.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399